Building An Evolution Transformation Library[1]

W. Lewis Johnson and Martin Feather

USC / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

We have been developing knowledge-based tools to support the evolutionary development of specifications. Evolution is accomplished by means of evolution transformations, which are meaning-changing transformations applied to formal specifications. A sizable library of evolution transformations has been developed for our specification language, Gist. This paper assesses the results of our previous work on evolution transformations. It then describes our current efforts to build a versatile, usable evolution transformation library. We have identified important dimensions along which to describe transformation functionality, so that one can assess the coverage of a library along each dimension. Potential applicability of this formal evolution paradigm to other environments will is assessed.

Keywords: specification, process modeling, evolution, intelligent assistance, component libraries

1. Introduction

The Knowledge-Based Software Assistant, as proposed in the 1983 report [13], was conceived as an integrated knowledge-based system to support all aspects of the software life cycle. Such an assistant would support specification-based software development: rather than writing code in conventional programming languages, programs would be written in an executable specification language, from which efficient implementations would be mechanically derived. A number of systems since been developed, each providing assistance for individual software activities: Sanders Associates' Knowledge Based Requirements Assistant [27], the Kestrel Institute's Performance Estimation Assistant [4], and ISI's Knowledge-Based Specification Assistant [17, 16, 15]. ISI and Sanders are now beginning development of a new system, ARIES,[2] which provides integrated assistance for requirements analysis and specification development.

ISI's work treats specification development as a formal evolutionary process. Requirements and specifications change rapidly during the requirements engineering process; this change must be supported and managed. In our approach a description of the system to be built exists in a machine-processable form from the very early stages of a software development project, and is gradually refined and evolved to produce a formal specification, together with supporting documentation. In ARIES, this system description is a mixture of formal representations and informal descriptions such as hypertext. This paper focuses only on the evolution of the formal component of system descriptions.

During the specification development process, a system description undergoes well-defined semantic changes [12]. New details are added, details of the domain are removed which are irrelevant to the system at hand, revisions are made to resolve conflicts between definitions, and high-level requirements on overall behavior are transformed into requirements on the behavior of individual system components. Evolution of requirements and specifications continues as a system is maintained. To support evolution, we have constructed a library of transformations for modifying specifications. This library consists primarily of so-called "evolution transformations", i.e., transformations whose purpose is to elaborate and change specifications in specific ways. They thus differ from conventional "correctness-preserving" transformations, which are applied to derive efficient implementations from specifications.

The original 1983 KBSA report anticipated that specifications would evolve, but did not describe the mechanism for such evolution. Partly as a result of the work on the Specification Assistant, the current vision of an ultimate Knowledge-Based Software Assistant embraces the notion of a formalized specification development process [7].

Our evolution transformations perform semantic changes such as revising the type hierarchy defined in a specification, changing data flow and control flow paths, and introducing processes to satisfy requirements. A single transformation may perform a number of

[1]This work was sponsored in part by the Air force Systems Command, Rome Air Development Center, under contracts F30602-85-C-0221 and F30602-89-C-0103. It was also sponsored in part by the Defense Advanced Research Projects Agency under contract no. NCC-2-520. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

[2]ARIES stands for Acquisition of Requirements and Incremental Evolution of Specifications.

0270-5257/90/0000/0238/$01.00 © 1990 IEEE

Recommended by: Dave Barstow

individual changes to a specification; for example, if a definition is changed, all references to that definition throughout the specification may be changed in a corresponding manner to retain semantic consistency.

Evolutionary development of specifications via transformations has the following advantages. First, because the transformations take care of low-level editing details, changes are performed more reliably. Second, a record of transformation steps provides traceability from high-level requirements to low-level specifications. Third, transformation steps can be undone and replayed. This latter capability is important in situations where multiple developers are independently making changes to a common specification. Specification consistency is maintained by taking each parallel sequence of evolution steps, attempting to combine them into a single sequence, and replaying the new sequence of steps, looking for conflicts between changes [9].

Exploration of the space of evolution transformations in the Specification Assistant project was example-driven. We concentrated on two problems, a patient monitoring system and an air traffic control system, and worked out development scenarios by hand to discover what transformations were necessary. We then implemented general-purpose versions of those transformations, which could be applied to achieve those developments mechanically. The result of this exploration was a sizable library containing around 100 transformations, of a wide variety of types. This library is significantly more extensive than similar libraries developed by Balzer [3] and Fickas [11]. And while other researchers have studied evolution steps similar to those captured by our transformations [28, 21], they have not developed transformations to enact these steps. We now are expanding on this work in the ARIES system, developing a transformation library that is extensive enough and powerful enough to apply to a wide range of specifications.

This paper summarizes our experiences so far with evolution transformations. We will compare evolution transformations and other representations of programming knowledge such as cliches. · Tools for applying and documenting transformations will be described. We will then describe our current work in building a proper library of transformations, and tools which aid in the use of this library. A scheme for categorizing transformations according to their effects will be described. This categorization system is driving the further development of the transformation library: it provides a map of the space of possible transformations, so we can determine where better coverage of this space by the library is needed. It also serves as the basis for automated tools which retrieve and apply transformations to accomplish the developer's desired semantic changes.

2. Related Work

We now compare our overall approach with related efforts toward supporting the *process* of specification construction. The following have influenced our research:

Burstall and Goguen argued that complex specifications should be put together from simple ones, and developed their language CLEAR to provide a mathematical foundation for this construction process [5]. They recognized that the construction process itself has structure, employs a number of repeatedly used operations, and is worthy of explicit formalization and support - a position that we agree with wholeheartedly.

Goldman observed that natural language descriptions of complex tasks often incorporate an evolutionary vein - the final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, etc., back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description [12]. He identified three "dimensions" of changes between successive descriptions: *structural* - concerning the amount of detail the specification reveals about each individual state of the process, *temporal* - concerning the amount of change between successive states revealed by the specification, and *coverage* - concerning the range of possible behaviors permitted by a specification. We were motivated by these observations about description to try to apply such an evolutionary approach to the *construction* of specifications.

Fickas suggested the application of an AI problem-solving approach to specification construction [10]. He identifies some domain-independent goals of specification construction, and methods to achieve them. Fundamental to his approach is the notion that the steps of the construction process can be viewed as the primitive operations of a more general problem-solving process, and are hence ultimately mechanizable. Continuing work in this direction is reported in [26] and [2]. They have concentrated on domain-specific goals arising in the course of specification development, whereas our efforts have concentrated on more problem-independent goals.

In the Programmer's Apprentice project (see [25] and, more recently, [30]), the aim - to build a tool which will act as an intelligent assistant to a skilled programmer - focusses on a different part of the software development activity to our work, yet much of what they have found has relevance to our enterprise. In their approach, programs are constructed by combining algorithmic fragments stored in a library. These algorithmic fragments are expressed using a sophisticated plan representation, with the resulting benefit of being readily combinable and identifiable. Use of the Programmer's Apprentice is thus centered around selection of the appropriate fragment and its composition with the growing program, with application of minor transformations to tailor these introduced fragments. In contrast, our approach is centered around selection of the appropriate evolution transformations, with reusable components playing a lesser role. Both efforts address issues of retrieval and explainability, and we will make specific comparisons later in the paper. Their more

recent project on supporting requirements acquisition (the "Requirements Apprentice", [24]) addresses the early stages of the software development process, and includes similar techniques to those of the Programmer's Apprentice but operating on representations of requirements.

3. Transformations in the Specification Assistant

In the Specification Assistant, evolution transformations were represented declaratively in such a way as to support interactive application, on-line documentation, and replay. Figure 3-1 shows the on-line documentation for the command Parameterize. This command is used to add parameters to parameterizable constructs in Gist, i.e., relations and events. The user supplies the parameter to be added to the definition. The command ensures that the type of the new parameter has been declared, and inserts it into the parameter list of the construct being parameterized. It then modifies every reference to the parameterized construct to add a new parameter there as well.

For each transformation, the Specification Assistant can generate on-line documentation, as shown above, as well as off-line documentation for inclusion in our reports and manuals. The system is also able to provide interactive help when transformations are applied. That is, when the user is trying to apply a Parameterize command, the system will guide the user through the application, indicating which parameters should be applied next, what their types are, and how they are to be input (e.g., "Please mark the the term to be parameterized (a parameterizable declaration). Mark with meta-Mouse-L."). The parameters are checked for validity, and then the transformation is applied.

A record is kept of each transformation that is applied, and on what arguments. The developer can undo a sequence of transformations at any time, perform additional changes, and then replay the transformations. The Assistant automatically determines whether the transformations are still applicable, and if not requests the developer to supply new inputs for the

```
Plan: PARAMETERIZE
    Add a parameter to a declaration, and attempt to generalize references to use new parameter.
Inputs:
    DECL: Term To be parameterized, a Parameterizeable construct;
              chosen by (Meta-LeftMouse)
    NEW-ROLE: New parameter, a role;
              chosen by (User typein)
Precondition:
    The new parameter type must be declared,
and there cannot be outstanding static analysis errors.
```

Figure 3-1: Documentation for the Parameterize command

transformations which could not be reapplied. This mechanism is used to simultaneously support evolution of the specification at different levels of detail, where one level is derived from the other by transformation. Two levels of detail which are useful to distinguish are the domain level and the system level. At the domain level, we would describe an air traffic control system in terms of its interactions with domain agents such as aircraft and controllers. At the system level, we would describe it in terms of actual inputs and outputs, such as radar tracks, flight plan data, and console displays. It is useful to be able to describe requirements in terms of domain objects, and then subsequently transform these into specifications of computations on system data. Replayability ensures that one can revise the domain-level description at will, and then rederive the system-level description. The replay capability can also be used to allow developers to elaborate different aspects of the specification independently. The developer tries one evolution sequence, undoes it, tries another, undoes it, and then replays the two sequences in an interleaved fashion.

3.1. The transformation representation

The following properties of transformations are made explicit in the representation. First, each transformation applies to a set of *inputs*, each of a particular type. Possible types are categories of syntactic constructs, such as expression or predicate, as well as disjunctions and specializations of these categories. These types are defined using ISI's AP5 database extension to Common Lisp [6]. In AP5, any predicate calculus query can be used as the definition of a new type, which can then be used in the definition of a transformation.

Transformations typically have a set of *preconditions*. Transformations typically have multiple preconditions. We attempt to specify preconditions for each transformation, sufficient to guarantee the applicability of the transformation. An interactive help facility in the Specification Assistant notifies the user which precondition failed, and requests the user to reenter those arguments which were used in the failed precondition. Preconditions are rechecked during transformation replay.

Finally, each transformation has a *method*. The method is an imperative program written in either our metaprogramming language Paddle [31], or Lisp. Once a transformation's preconditions are satisfied, the method is executed, and the result is displayed.

Additional textual properties were included in order to support the documentation and help facilities. Each transformation is given a short phrase to use as the name of the transformation, and a longer string explaining what the transformation does. Likewise English phrases

are supplied for each input, type, precondition, and precondition failure. These phrases are assembled to provide the English text necessary for each interaction with the user.

3.2. Transformation categories

We divided our set of transformations into the following categories, to facilitate users' selection of the the appropriate transformation:

- Structure-adding commands, which add a new construct into the specification, e.g., Add-Type, which adds a new type declaration,
- Replacement commands, which replace a construct with a new one, e.g., Rename-Concept, which changes the name of a definition,
- Reorganizing commands, which restructure the specification without changing its meaning, e.g., Bubble-Up, which moves a definition up out of an enclosing module,
- Behavior changing commands, which modify the behavior described by existing specification components, e.g., Singleton-To-Any, which permits multiple instances of a type where only a unique instance was permitted before,
- Data flow modifying commands, which change the flow of data through the system, while retaining the behavior of the system as a whole, e.g., Spice-Communicator, which interposes a communication device between two agents,
- Terminology elaboration commands, which elaborate some part of the specification terminology, often by adding or changing an existing declaration,
- Unfolding commands, which replace uses of a construct with equivalent but lower-level constructs, e.g., Unfold-Function, which changes a function definition into an equivalent relation definition,
- Implementation / approximate unfolding commands, which replace uses of a construct with a nearly equivalent construct which is closer to implementation, e.g., Maintain-Invariant-Reactively, which replaces an invariant with a demon which reacts to violations of the invariant, and
- Abstracting commands, which make a specification more abstract by discarding detail, e.g., Collapse-Types, which combines two types into a single more general type.

4. Analyzing Transformation Effects

The above categories only characterize the effects of transformations in general terms. In our current work, we have been attempting to formalize more precisely the semantic changes that transformations achieve. We are now incorporating these formal descriptions of effects into the command definitions, to further aid in retrieval and application. We are also extending the library to ensure that the space of possible semantic changes which we have identified is adequately covered.

There are two main types of transformations in our library, those which change meaning and those which do not. The "meaning-preserving" transformations differ from those appearing in transformational implementation systems in intended purpose: they

- reorder the specification components for better presentation,
- rewrite specification components into an equivalent form using different language constructs, or
- eliminate redundancies and make explicit some otherwise implied features of the specification.

Some of these transformations may also appear in a transformational implementation system, since such systems must frequently replace high-level constructs with low-level ones.

To describe the meaning-changing transformations, a language is needed for describing the semantic properties being changed, and the nature of the change. In many cases, it is natural to view the specification as a semantic network, and the transformation as a modification of that network. Network notations such as entity-relation diagrams and data flow diagrams are commonly used in describing systems. Each such diagram focuses on one dimension of the system. Evolution transformations have the effect of modifying one or more of these network diagrams. Parameterize, for example, changes the entity-relationship model: it adds a new link between the concept being parameterized and the type of the new parameter. If the construct being parameterized is a process definition, Parameterize also changes the data-flow network: it adds an input to the process. We have identified a number of such network abstractions, and have developed tools for deriving each abstraction from system specifications. The effects of transformations can be characterized using a set of generic network modification operators, which can be applied to any dimension.

The syntactic constituents of the specification are used to represent the nodes in the semantic networks. Every declaration is a node, as is every parameter of the declarations, every statement, procedure, and expression. Furthermore, every object is linked to the syntactic subconstituents that it contains, and to the constituent

that contains it. A parse tree is thus a natural representation for such a network of nodes.

A range of abstract semantic diagrams can be derived from the basic parse tree network. This involves selecting the nodes that are included in the abstract network, and defining links between them in terms of the structural relationships of the parse tree. An entity-relationship diagram, for example, only includes the type, instance, relationship, and event nodes; it omits nodes which are internal to the definition of each. Abstract semantic relations can be defined in terms of co-occurrences of syntactic relations. For example, an *invokes* relation holds between a procedure call and a procedure if the procedure appears syntactically within the lexical environment of the procedure call, and has the same name as the name appearing in the procedure call. In Gist, the type hierarchy is defined using `specialization-of` slots, as in the following example: `type pilot specialization-of person`. A *specialization-of* relation can be defined to exist between two types if the declaration of the supertype is in the lexical environment of the declaration of the subtype, and the subtype definition has a `specialization-of` slot whose value is the name of supertype.

The following semantic relationships are effective for describing changes in the transformations that we have studied so far:

- the modular organization of the specification, represented as a `component` relation between modules and their components,
- the entity-relationship model defined in the specification, represented using the relations `specialization-of`, and `parameter-of`, `type-of`, and `instance-of`,
- the data flow relationships, holding between components that computed values and components that supply values,
- control flow links, consisting of the relationships `control-substep` and `control-successor`,
- fact flow links, consisting of the relationships `accesses-fact` and `modifies-fact` between processes and the declarations of facts that they access and modify, and
- state description links, associating statements and events, on one hand, with preconditions and postconditions that must hold in the states before and after execution, respectively.

The distinction between data flow and fact flow arose out of our attempts to reconcile conventional data flow representations with the relational world model embodied in Gist. In the data flow view, processes interact with the environment only through fixed input and output ports. In the relational modeling view, facts about the environment are represented as a database of assertions that processes can freely access and modify. We believe that strict adherence to data flow through input and outputs is inappropriate in high-level specifications, because it confuses the concerns of what behavior is desired and what data will flow between processes to accomplish this behavior. For example, we may wish to state in a specification for an air traffic control system the assumption that "all aircraft of interest are flying below 30,000 feet". Although this assumption refers to the aircraft's altitude, altitude data will be input to the process only if the process explicitly checks the assumption at run time. Fact flow refers to the properties of domain objects which a process is sensitive to, or which are changed by the process, regardless of whether these properties are manifested as data flows. In our specification development methodology, developers are free to first define the relevant fact flows and then use evolution transformations to turn these into data flows.

The most primitive network manipulation operations are `insert` and `remove` for adding and deleting links, and `create` and `destroy` for creating and destroying objects. In addition to these primitive operations, a number of complex operations have been identified:

- `Update` - remove a link from one node and add it to another node.
- `Promote` - a specialization of `update`. If one of the linked nodes is part of an ordered lattice, then update the link so that it connects a higher node in the lattice.
- `Demote` - the opposite of `promote`. Move the link to a lower node in the lattice.
- `Splice` - remove a link from between two nodes A and B, and reroute the connection through a third node, C, so that A is linked to C and C is linked to B.

Some examples of `promote` transformations are the following:

- Bubble-Up - move a definition from an embedded module to an enclosing module - this promotes over the `component` hierarchy;
- Generalize-Parameter - change the type of a parameter to be more general - this promotes over the `specialization-of` hierarchy.

Note that for some of these transformations it may be necessary to specify over which link the promotion or demotion is supposed to occur.

`Splice` arises in a number of different transformations, splicing a range of different links. For example, Splice

Communicator splices **accesses-fact** paths. It is applied to some process definition that is accessing some fact about objects in the external world. The transformation introduces a new object, typically a monitoring device, to serve as an intermediary. The new object accesses the external object, and copies the accessed information locally. The process can then access the copied data instead of the external data. Splice Communicator is thus a common step in transforming abstract fact flows into concrete data flows. Other transformations perform splicing operations, particularly those which fold in new definitions. One such transformation is Statement-To-Procedure, which takes a statement which is part of some process definition, and defines it as a separate procedure. The statement is then executed by invoking the new procedure. Viewed from the control-flow perspective, Statement-To-Procedure splices a procedure invocation between the caller and the new procedure body. This analysis reveals an interesting relationship between splicing and folding. Folding a definition has the effect of drawing a boundary around a section of text, and turning it into a new definition. Any links to the text must therefore be spliced in order to get across the new boundary.

We have also identified some more complex object creation and destruction operations:

- **Join** - combine two nodes into a single node; some links to the rest of the network may be lost.
- **Copy** - Construct a new node as a copy of an existing node; some links may not be copied over.

5. Building a More Complete Library

The above analysis has given us the means for determining what transformations to include in our library, and the basis for assessing its completeness. It helps to identify "building-block" transformations from which more complex transformations may be constructed. Building-block transformations perform some simple operation along one or more dimensions, rather than performing multiple operations along a single dimension, or multiple unrelated operations. Our previous example-driven approach led to transformations that that were overly complicated, even when described in our dimensional fashion. These transformations need to be decomposed into their component building blocks.

An example of such a transformation is Add-Disjoint-Subtypes, first described by Balzer [3]. This transformation defines two disjoint subtypes of a specified type, and revises the signatures of all relations over the type so that they instead are restricted to one subtype or the other. The operations of adding the subtypes and specializing the relations are distinct operations, which could be performed independently of each other. We therefore realized individual operations in separate transformations, Add-Specialization and Specialize-Parameter. Add-Disjoint-Subtypes now invokes these other transformations as substeps. Users are free to either use the larger command, or invoke the substeps directly for some other purpose.

Coverage of the library can be assessed by making sure that all possible combinations of network update operations, link classes, and node classes are represented in the library. This analysis has already resulted in improved coverage. For example, once **splice** was recognized to be a generic operation applicable to a variety of links, it could be applied to other links such as control flow. This led to the inclusion of the Statement-To-Procedure transformation.

In our framework, specification objects belong to a hierarchy of types, and relations belong to a hierarchy as well. The effect descriptions of transformations may be more specific or general, depending upon the generality of the object types and relations involved. One way of identifying opportunities for new transformations is to see whether a transformation can be modified to apply to a more general class of objects, or conversely whether a more powerful transformation should be written which applies to a narrower class of objects.

We are already well on our way to providing a firmer foundation for our transformation library, using more general transformation building blocks. However, the resulting library may turn out to be inadequate for a number of reasons. First, there may be some links between nodes that we have failed to identify. As we broaden our specification language to include more non-functional requirements, new dimensions will likely emerge. Second, there may be higher-level operations which we have failed to identify, i.e., other operations such as **splice**. If we discover a new generic operation, then a whole new category of transformations will then be suggested.

Previous work in the development of reusable component libraries has generally been unable to provide methods for assessing library completeness along the lines identified here. For example, Prieto-Diaz and Freeman [22] factor software component properties along a number of different dimensions, but fail to establish a taxonomic hierarchy along any dimension. There is thus no way to determine whether one component is more general in applicability than another. Systems which rely upon a classification hierarchy of objects, such as Allen's system [1] and Lubars' system [19] classify primarily on the basis of input and object types. Without a notion of generic operations, classification of effect is *ad hoc* at best. By restricting our classification to a particular kind of software component, namely evolution transformations, we are able to do a much better job of classifying our components.

6. Retrieving and Using Transformations

When using a library of operators such as our evolution transformation library, a user must do the following:

- find the right transformation,
- understand what it does, and
- determine how to apply it to achieve the desired effect.

These activities are seldom trivial, particularly if the population of the library is large and the effects of the operators is potentially complex. We have consequently been concerned with providing automated support for these activities. Some new capabilities in these directions have recently been developed, and further developments are anticipated.

Instead of relying upon fixed menus, we provide a flexible retrieval mechanism for identifying candidate transformations to apply. Each transformation is annotated to indicate what effect it is guaranteed to have, and what effects is may possibly have under certain circumstances. These annotations are lists of effect descriptions, along the lines described above. Each effect is a generic operation applied to a combination of the transformation's inputs, outputs, and other related objects which are not directly input or output. The user can then specify a desired effect in terms of the class of operation, and the objects of interest. For each operand an object class may be specified, or a particular specification component object may be referred to. Given this description, the retrieval mechanism retrieves three sets of transformations:

- those which are guaranteed to achieve the desired effect,
- those which may achieve the desired effect, but only in restricted circumstances, and
- those which achieve part of the desired effect.

We see this capability as the kernel of a retrieval system which would help users iteratively revise their effect descriptions in order to locate the right transformation. Now that preconditions and effects of each transformation are made explicit, it will be possible for ARIES to suggest additional transformations to apply to correct precondition failures. If a transformation achieves part of the desired effect, it will be possible to search the transformation library for transformations that complete the evolution step. With these capabilities transformation application will become more of an interactive planning process, and the ARIES system will become a more active assistant in the specification development process.

7. Examples

To convey a flavor of the contents our library of transformations and how to use it we show some small examples of semantic relationships, effects and transformations.

7.1. Manipulations and the semantic relationships

One of the major themes of this paper is that our transformation library can be organized and accessed via the *effects* induced by transformations, and that such effects can be represented as network manipulation operations on semantic relationships. For example, consider one such manipulation operations, *splice*. As discussed earlier, *splice* removes a link between two nodes A and B, and re-routes the connection through a third node, C, so that A is linked to C and C is linked to B - figure 7-1

```
Splicing changes link:    A --> B
                    to:   A --> C --> B
```

Figure 7-1: Splicing a link

The kind of effect achieved by splicing depends upon the kind of semantic relationship represented by the link being spliced. Table 7-1 summarizes the possibilities for the relationships that we have studied.

7.2. Accessing transformations in the library

To find the transformation(s) that manipulate a particular kind of semantic relationship, we issue a retrieval query which expresses the nature of the effect we are seeking. Such a query must state:

- the network manipulation operation, one of: insert, remove, update, promote, demote, or splice,
- the semantic relationship to be manipulated by the operator, one of: control flow, data flow, etc., and
- the argument(s) to the operation: either a reference to an actual piece of specification, or a generic description - "a statement", "a type-declaration", etc.

For example, if we wish to find the transformations that splice a control-flow substep link between two statements, we could issue a retrieval query with

```
operation - splice
semantic relationship - control-subste
arguments - "a statement", "a statemen
```

Similarly, if we wish to find the transformations that add a type declaration to a module, we could issue a retrieval query with

```
operation - insert
semantic relationship - component
arguments - "a module", "a type-declar
```

Currently we have only a textual-level program interface for exercising this retrieval capability. It is our intention to use a graphical display of nodes and links in conjunction with menus of operations as the more appropriate user interface.

control flow

MEANING OF A LINK "S1 --> S2":
control flows from statement S1 *to* S2.
EXAMPLES
splicing in sequential composition:
steps[S1; S2] *becomes* steps[S1; S3; S2]
effect on control flow links:
S1 --> S2 *becomes* S1 --> S3 --> S2

splicing in substatements:
steps[S1] *becomes* steps[if P then S1]
effect on control flow links:
steps[S1] --> S1 *becomes*
steps[if P then S1] --> if P then S1 --> S1

data flow

MEANING OF A LINK "E1 --> E2":
data flows from expression E1 *to* E2.
EXAMPLE
splicing in nested expressions:
F(x,?) *becomes* F(G(x,?),?)
effect on data flow links:
x --> F(x,?) *becomes* x --> G(x,?) --> F(G(x),?)

e-r model

MEANING OF A LINK "C1 --> C2":
concept C1 *is a specialization of* C2.
EXAMPLE
splicing in type hierarchy:
TYPE T1 specialization of T2 *becomes*
TYPE T1 specialization of T3;
TYPE T3 specialization of T2
effect on specialization-of links:
T1 --> T2 *becomes* T1 --> T3 --> T2

modular organization

MEANING OF A LINK "C --> M":
component C *belongs to module* M.
EXAMPLE
splicing in modular inclusion:
module M1 internal { relation R() } *becomes*
module M1 internal
{ module M2 internal { relation R() } }
effect on belongs to links:
R --> M1 *becomes* R --> M2 --> M1

state description

These links are not "spliceable".

Table 7-1: Splicing the different semantic relationships

7.3. Splicing transformations in action

Finally, we illustrate the use of a transformation that splices the data flow link. Suppose that we are focusing on the following fragment of a specification:

```
RELATION P(x,y) IFF y = IP(x,?)
```

This defines a binary relation P to hold between two arguments x and y if and only if y equals IP(x,?) i.e., some value which in the place of the "?" makes IP(x,?) true. (The "?" notation is our way to retain a relational model while getting some of the convenience of a functional style of expressions. Equivalently, we could have written

```
... IFF IP(x,y)
```

which means the same, but has a less explicit data flow.)

We might wish some post-processing to be done on the value returned by the call on IP. To insert such processing, we use an evolution transformation to *splice* it into the data flow from IP(x,?) to y. In particular, we may use our SPLICE-AROUND-EXPRESSION transformation, giving it as inputs:

```
where to do the splicing:
      around the retrieval of the value from IP, i.e., I

what to be spliced in:
      POST(v,?), where the original expression goes in
```

The result of this transformation is:

```
RELATION P(x,y) IFF y = POST(IP(x,?),?)
```

Alternatively, if we wanted to do some pre-computation on x to reject some inputs, we could use the same transformation to splice this computation into the data flow from P's input argument x to the predicate y = POST(IP(x,?),?), i.e., we would give the transformation the inputs:

```
where to do the splicing:
      around the predicate that defines P

what to be spliced in:
      PRE(x) and w, where the original predicate goes
```

The result of this transformation (applied to the original program fragment, *not* the one into which post-processing has been introduced) is:

```
RELATION P(x,y) IFF PRE(x) and y = IP(x,?)
```

Suppose that we wished to have *both* evolutions, i.e., pre-processing to reject some x inputs, and post-processing of the results returned by IP. To achieve this we wish to perform the two evolution transformations in series. Knowing the effect of these transformations and where they apply, namely that they each splice a data flow link, but the spliced links are *different* links of the original program fragment, we can deduce that both evolution transformations can be serially applied without conflict, to result in the following:

```
RELATION P(x,y) IFF PRE(x) and y = POST(IP(
```

Conversely, had the two transformations been splices of the *same* data flow link, we would have recognized a conflict if both were to be performed. This kind of reasoning - making use of knowledge of the applications of evolution transformations, and the effects of those transformations - is necessary to realize the kind of specification development suggested in [8].

8. Changing the Transformation Representation

We will now summarize the changes to the representation of transformations, reflecting our improved understanding of the their effects. This changes allow for more consise descriptions of transformation methods and effects, and provide a

stronger foundation for interactive application and explanation of transformations.

As suggested above, our transformations perform changes to specifications via syntactic structure modifications on parse trees. The complexity of the transformation methods depends upon how easy it is to derive semantic properties from syntactic structures. The syntax of Gist is very complex, with a number of equivalent alternative syntactic constructs and an idiosyncratic syntactic form. These syntactic features make transformations overly complex. Accordingly, we defined a new grammar, with a regular, attribute-value form, and a minimum number of constructs; this grammar is used internally by the system. We have found that the transformations operating upon this internal form are considerably simpler than their equivalents operating upon the old parse tree representations. Furthermore, fewer meaning-preserving transformations are required, because there are fewer alternative forms to transform between. Translators are employed to convert between the old surface syntax for Gist and the internal grammar. Thus the surface syntax becomes merely a presentation of the underlying system description, one of many possible presentations. We plan to construct translators from the internal representation to graphical presentations as well. This will enable the user to directly manipulate the appropriate graphical network abstraction when describing a desired evolution step.

Another major change has been to implement transformation methods directly as sequences of updates to a semantic networks. In the Specification Assistant, transformation methods employed the Popart Editor to make specification changes; this editor would either be called from Paddle or Lisp. The Popart Editor operates by walking the parse tree, maintaining a pointer into the parse tree which determines the current editor focus. Changes are made to the parse tree by deleting, replacing, or inserting next to whatever the current focus is. Such methods are difficult to analyze from the standpoint of network updates because a) a substantial portion of a transformation method is devoted to walking the parse tree instead of making changes; b) determining the effect of a change requires simulating the parse tree walk to determine where the editor focus will be; c) parse tree traversal can only follow links between nodes in the parse tree. The solution of these problems was to make the parse tree manipulatable as a semantic database. A package called Popart-DB was developed which allows one to view parse trees as a database of relations in the AP5 database system [18]. The structural links of the parse tree are then represented explicitly as relations between database objects, and the transformations make insertions, deletions, and updates of these relations. Thus the primitive operators in transformations are the same as the primitive operators used to describe transformation effects. Furthermore, the abstract relations between specification component described in Section 4 can be defined in terms of these base-level relations. This allows the transformation methods to traverse these abstract relations. In some cases update methods are defined for the abstract relations as well, so that the transformations can update them directly. Thus the transformation methods and the transformation effect descriptions are defined in much the same language, as is commonplace in most AI planning systems, such as Grapple [14].

The Gist language models the world as collections of facts, and models behavior in terms of changes to facts in the world. It thus models the world as a database. The natural next step was to start using Gist as the language for writing transformations, or at least the subset of Gist that is automatically compilable into AP5. At the time of this writing we have converted about half of our transformations into Gist. Advantages stem from the increased uniformity - tools that operate upon Gist (e.g., for explanation and analysis) can now be applied to the transformations themselves. One such tool is the Gist Paraphraser [29, 20], a facility which generates natural language descriptions of Gist specifications. Use of the Paraphaser in describing evolution transformations will improve understandability of the transformations, with less reliance on canned documentation. Furthermore, the same tools which are developed to analyze the behavior of Gist specifications will be usable to determine the cumulative effects of transformation methods. This will allow transformation effect descriptions to be largely determined automatically, rather than being manually annotated as is currently done.

Figure 8-1 shows the new representation of Parameterize.[3] This example illustrates the above points, as well as some additional ones listed below.

We have started including with the transformations examples of their use. A small example specification, and an example command invocation, are supplied. These aid in on-line documentation of the transformations; they also assist in testing and validation of the transformation library. In addition, hypertext documentation is included in the representation. Italicized words in the documentation strings are names of related knowledge base concepts, which can be browsed by the user.

We explicitly represent the effects of transformations. For each transformation, we explicitly record its effects in terms of the network operations listed earlier. We also identify possible side effects. The operands of these operations may be inputs to the transformations, outputs of the transformation, or some other objects retrieved from the database. Our previous representation made inputs explicit, but did not make outputs explicit. The new representation makes both inputs and outputs

[3]Note that we have made some minor syntactic changes to Gist, to make it somewhat closer to REFINE and related languages. We use ":" to mean "of type" now, "." for attribute retrieval, and "|" to mean "such that".

Transformation: Parameterize

Concept description: "Add a new *parameter* to a *concept-declaration*, and modify references to include references to the *added parameter*."
Parameters:
Form: `decl : concept-declaration`
Display Name: "*concept-declaration* to be parameterized"
Form: `param-name : symbol`
Display Name: "Name of the *added parameter*"
Form: `param-type : type-declaration`
Display Name: "type of the *added parameter*"
Form: `expression : expression-tree`
`default dummy-actual(new-parameter)`
Display Name: "The new *actual*"
Concept Description: "An *expression* to be inserted into references to the *concept-declaration*, to compute the new actual"
Notes: "The *expression* may use the names of existing formal *parameters* freely to refer to their corresponding actual values"
Outputs:
Form: `added-parameter : parameter`
Display Name: "added *parameter*"
Precondition: `not exists x:parameter | (parameter-of(decl, x) and name(x, param-name))`
Failure Message: "A *parameter* named `[param-name]` already exists"
Concept Description: "A *parameter* of the same name must not exist"
Method:

```
steps [add-parameter-to-signature[new-parameter, decl]
   yielding added-parameter;
      add-actual-computed-wrt-existing-actuals[decl,
      expression]]
```

Example Spec:

```
{type aircraft;
 relation controlled(ac : aircraft);
 relation in-flight(ac : aircraft);
 invariant foo for-all ac : aircraft |
   in-flight(ac) => controlled(ac)}
```

Example Invocation:

```
parameterize[declaration-of('controlled,
             'relation-declaration),
             gist-template("c : controller"),
             declaration-of('controller, 'type-declaration),
             gist-template("ac.assigned-controller")]
```

Figure 8-1: New definition of Parameterize

explicit, to facilitate the definition of effects. At the present time all possible side effects must be explicitly recorded, to support all relevant retrieval queries. However, since the semantic relations are formally derived from syntactic features of the specification, it will be a straightforward matter to derive possible side effects from transformations in a similar fashion.

9. Applicable Results and Future Challenges

Formalized evolution transformations are a potential benefit to all software evolution activities, not just specification development. The analysis of transformations in this paper provides a framework for applying evolution transformations to other languages. Any language which supports the mechanical derivation of semantic relations on software objects is a candidate for formalized evolution. Strongly typed languages such as Ada and Pascal fit into this category. Unrestricted Lisp is a poor candidate because it does not rely much on typing mechanisms, and because dynamic scoping of variables makes it difficult to trace data flow through a program.

The main technical prerequisite to our evolution approach is a knowledge representation framework that supports retrieval and updates of the relevant program features in an abstract form. We rely heavily upon our AP5 and Popart-DB tools; other frameworks such as Refine [23] can provide similar capabilities. Without suitable abstractions, transformations are difficult to write and understand, and are unlikely to be versatile enough to support the construction of a good library.

This paper has described the semantic basis for developing a reusable library of transformations. Work on extending the coverage of the library is ongoing. The main technical challenges that remain have to do with providing sufficient automated support for the transformation retrieval and application processes, and for deriving effects and preconditions of transformations from their method bodies. Given the work accomplished so far, we believe that it will be straightforward to develop a system which retrieves transformations through iterative reformulation of queries, as in BACKBORD [32], and which guides the user in applying the transformation to achieve the desired effect. We envision that the system ultimately will take a more active role in interactive planning of specification changes. Failed preconditions on transformations could then trigger a search of the transformation library for transformations that could make the preconditions true. The system could provide suggestions at each stage as to what transformations would be appropriate to perform.

10. Acknowledgements

We wish to acknowledge the member of the KBSA project at ISI for their participation in this research, in particular Jay Myers, Dan Kogan, Kai Yue, and Kevin Benner. We also wish to thank Jay Myers and K. Narayanaswamy, who reviewed earlier drafts of this paper.

References

1. Allen, B.P., Holtzman, P.L., and Lee, S.D. A Knowledge-Based Environment for the Development of Software Parts Composition Systems.

2. Anderson, J.S. & Fickas, S. A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May, 1989, pp. 177-184.

3. Balzer, R. Automated Enhancement of Knowledge Representations. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, AAAI, August 18-23, 1985, pp. 203-207.

4. Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., and Westfold, S. Progress on the KBSA Performance Estimation Assistant. Proceedings of the 2nd KBSA Conference, 1988.

5. Burstall, R.M. & Goguen, J. Putting theories together to make specifications. Proceedings of the Fifth International Conference on Artificial Intelligence, August, 1977, pp. 1045-1058.

6. Cohen, D. *AP5 Manual.* USC-Information Sciences Institute, 1985. Draft.

7. Elefante, D. Overview of the Knowledge Based Specification Assistant. Proceedings of the Computers in Aerospace VII Conference, Monterey, CA, 1989.

8. Feather, M.S. "Constructing Specifications by Combining Parallel Elaborations". *IEEE Transactions on Software Engineering 15*, 2 (February 1989), 198-208. Available as research report # RS-88-216 from ISI, 4676 Admiralty Way, Marina del Rey, CA 90292.

9. Feather, M.S. Detecting interference when merging specification evolutions. Accepted for the 5th International Workshop on Software Specification and Design, May 1989.

10. Fickas, S. A Knowledge-Based Approach to Specification Acquisition and Construction. Tech. Rept. 86-1, CS Dept., University of Oregon, Eugene, 1986.

11. Fickas, S. Automating the Specification Process. Tech. Rept. CIS-TR-87-05, Department of Computer and Information Science, University of Oregon, 1987.

12. Goldman, N. M. Three Dimensions of Design Development. Tech. Rept. Information Sciences Institute/RS-83-2, USC-Information Sciences Institute, July, 1983.

13. Green, C., D. Luckham, R. Balzer, T. Cheatham, C. Rich. Report on a Knowledge-Based Software Assistant. In Rich, C., and Waters, R., Ed., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, Los Altos, CA, 1986.

14. Huff, K.E., and Lesser, V.R. The GRAPPLE Plan Formalism. Tech. Rept. 87-08, U. Mass. Department of Computer and Information Science, April, 1987.

15. Johnson, W.L. Deriving specifications from requirements. Proceedings of the 10th International Conference on Software Engineering, 1988, pp. 428-437.

16. Johnson, W.L. Specification as Formalizing and Transforming Domain Knowledge. Proceedings of the AAAI Workshop on Automating Software Design, 1988.

17. The KBSA Project. Knowledge-Based Specification Assistant: Final Report.

18. Johnson, W.L., and Yue, K. An Integrated Specification Development Framework. Tech. Rept. RS-88-215, USC / Information Sciences Institute, 1988.

19. Lubars, M., and Harandi, M. Addressing Software Reuse through Knowledge-Based Design. In Biggerstaff, T.J., and Perlis, A.J., Ed., *Software Reusability*, Addison Wesley, 1989.

20. Myers, J.J., and Johnson, W.L. Towards Specification Explanation: Issues and Lessons. Proceedings of the 3d Knowledge-Based Software Assistant Conference, 1988.

21. Johnson, P. Structural Evolution in Exploratory Software Development. Proceedings of the AAAI Spring Symposium on Software Engineering, 1989.

22. Prieto-Diaz, R., and Freeman, P. "Classifying Software for Reusability". *IEEE Software 5*, 1 (January 1987).

23. *Refine User's Guide.* Reasoning Systems, Palo Alto, CA, 1986.

24. Reubenstein, H.B. & Waters, R.C. The Requirements Apprentice: An Initial Scenario. Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May, 1989, pp. 211-218.

25. Rich, C., Schrobe, H.E. & Waters, R.C. An overview of the Programmer's Apprentice. Proceedings, 6th International Joint Conference on Artificial Intelligence, 1979, pp. 827-828.

26. Robinson, W.N. Integrating Multiple Specificationss Using Domain Goals. Proceedings, 5th International Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, May, 1989, pp. 219-226.

27. Sanders Associates. Knowledge-Based Requirements Assistant - Interim technical report. Software Systems Engineering Directorate, March, 1986.

28. Narayanaswamy, K. Static Analysis-Based Program Evolution Support in the Common Lisp Framework. Proceedings of the 10th International Software Engineering Conf., 1988.

29. Swartout, W. GIST English Generator. Proceedings of the National Conference on Artificial Intelligence , AAAI, 1982.

30. Waters, R.C. "The programmer's apprentice: A session with KBEmacs". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1296-1320.

31. Wile, D.S. Program developments: Formal explanations of implementations. In *New Paradigms for Software Development*, IEEE Computer Society Press, 1986, pp. 239-248. Also published in *CACM* 26, (11), 1983, 902-911..

32. Yen, J., Neches, R., and DeBellis, M. Specification by Reformulation: A Paradigm for Building Integrated User Support Environments. Proceedings of the Seventh National Conference on Artificial Intelligence, AAAI, 1988.